Codeon: On-Demand Software Development Assistance

Yan Chen[1], **Sang Won Lee**[2],**Yin Xie**[1], **YiWei Yang**[2], **Walter S. Lasecki**[2,1]**, and Steve Oney**[1,2]

School of Information[1], Computer Science & Engineering[2], University of Michigan, Ann Arbor
{yanchenm,snaglee,xieyin,yanyiwei,wlasecki,soney}@umich.edu

ABSTRACT

Software developers rely on support from a variety of resources—including other developers—but the coordination cost of finding another developer with relevant experience, explaining the context of the problem, composing a specific help request, and providing access to relevant code is prohibitively high for all but the largest of tasks. Existing technologies for synchronous communication (e.g. voice chat) have high scheduling costs, and asynchronous communication tools (e.g. forums) require developers to carefully describe their code context to yield useful responses. This paper introduces Codeon, a system that enables more effective task hand-off between end-user developers and remote helpers by allowing *asynchronous* responses to on-demand requests. With Codeon, developers can request help by speaking their requests aloud within the context of their IDE. Codeon automatically captures the relevant code context and allows remote helpers to respond with high-level descriptions, code annotations, code snippets, and natural language explanations. Developers can then immediately view and integrate these responses into their code. In this paper, we describe Codeon, the studies that guided its design, and our evaluation that its effectiveness as a support tool. In our evaluation, developers using Codeon completed nearly twice as many tasks as those who used state-of-the-art synchronous video and code sharing tools, by reducing the coordination costs of seeking assistance from other developers.

Author Keywords

Development support; intelligent assistants; crowdsourcing

ACM Classification Keywords

H.5.m Information Interfaces and Presentation (e.g. HCI): Miscellaneous; K.6.1 Management of Computing and Information Systems: Software Development

INTRODUCTION

Software developers rely heavily on support from external resources while programming. Although search engines and Community Question-Answering (CQA) websites (such as

StackOverflow [33]) are the most popular resources for developers, the best support is often provided by other developers [31, 20, 12]. Unlike web-based resources, expert developers can provide personalized help, high-level advice, and project-specific code segments, and can often help identify and overcome bugs that are difficult for a single developer to find on their own [35]. However, it is often prohibitively difficult to find other developers willing to help, particularly for developers working outside of a large organization.

Recently, a small set of paid services began connecting developers with remote expert developers [23, 21], who can provide personalized feedback. These services use a *synchronous*, one-on-one model of communication where developers connect to a remote expert, make a request, and communicate via video chat and a shared editor. However, there are several drawbacks to this synchronous model [12]. There is a coordination cost of finding an expert who is available to help at the right time. If the first expert does not have sufficient expertise (which they cannot know until *after* they connect), there is a further cost—in both time and money—to finding a new expert. One-on-one mentoring also requires that the developer be attentive to the remote helper throughout the session. Although this is suitable for teaching-oriented requests where a back-and-forth conversation between developers and helpers is desirable, it is less helpful for tasks that can be handed off entirely to the helper, such as requests for short code snippets.

On-Demand Programming Assistance with Codeon

In this paper, we propose an *asynchronous* on-demand help seeking model for programmers who need support that can be more efficiently provided by remote expert developers than existing methods. We implement and evaluate this model in Codeon, a system that allows developers to request assistance as easily as they can through in-person one-on-one communication, and tracks helpers' responses, directly in the developer's Integrated Development Environment (IDE). As we will show, Codeon makes remote collaboration more practical by reducing coordination costs while still enabling rich communication between developers and helpers. Unlike previous asynchronous collaboration solutions (such as code repositories), Codeon is request-oriented: it makes it easy for developers to make sufficiently detailed requests and send to other developers, making the process quick and effective. Further, Codeon's asynchronous model is more scalable for multiple helpers than synchronous support tools because it allows multiple helpers to work in parallel with the developer. As our evaluation demonstrates, Codeon supports new forms of par-

allel collaboration that make remote help-seeking more effective for developers. In this paper, we contribute the following:

- an effective approach for integrating external, parallelizable expert assistance into a developer's on-going process,
- tools and techniques that efficiently capture developers' requests' contexts and mediate communication between end users and remote developer helpers,
- evaluations of the trade-offs between speed and accuracy for system components in different help seeking stages,
- a system (Codeon) that instantiates our approach to improve development help-seeking tools, and
- evidence that Codeon helps developers solve more tasks in a given time span than current approaches.

We begin this paper with a discussion of related work. We then discuss how we designed and evaluated our system, followed by limitations and future work.

RELATED WORK

Our work builds on previous research into pair programming, developer help-seeking, distributed programming, and communication support tools.

Help Seeking in Software Development

Community Question Answering

Many CQA websites, such as Stack Overflow [33], provide online asynchronous support that allows software developers to post questions to a large community. These sites also accumulated these questions and answers that they received to form a large database of questions and answers for later reference. Prior work [2, 3] studied building an "organizational memory" through a growing database of questions and answers. These CQA sites have a number of limitations such as long waiting times to receive an answer after posting the requests and the face that answers are not personalized. In addition it takes significant time and effort to compose a question with enough context and explanation for other programmers to be able to provide an answer [5]. Codeon enables speech and content selecting modalities, and also provides on-demand expert support that allows developers to describe the requests as if the helpers were physically nearby. They can select a code snippet, verbally ask "what does this mean?", hand off execution of planned coordination to the system, and receive a meaningful response within minutes.

Commercial support platforms, such as Code Mentor [21] and hack.hands() [23], provide more personalized help for software developers. These sites allow developers to create requests and connect them (or let them self-select) with experts, and provide a shared code editor and text/voice communication channel. These sites represent the state of the art for seeking remote help from experts and use synchronous one-on-one communication. In our system evaluation, we show that on-demand support yields similar one-on-one support results while also having the benefit of being parallelizable.

Pair programming

Codeon is related to pair programming [14], a method that allows developers to work together in real-time more effectively. In particular, it is most related to distributed pair programming, which is a derived version of pair programming,

allows remote participants to contribute to the same codebase simultaneously [6, 38]. Although the distributed pair programming approach removes many issues in real-time remote collaboration [32], it can still be difficult to coordinate and maintain context in distributed pairs. Our system instead aims to automate coordination by temporarily incorporating helpers into a task long enough for them to assist and then move on.

Information Needs for Developers

Researchers have summarized the types of questions that developers ask in different contexts. Sillito et al. categorized 44 types of questions developers ask when evolving a large code base [39]. Ko et al. explored six types of learning barriers in programming systems for beginners and proposed possible solutions from programming system sides [26], and also documented communication among co-located development teams [25]. Guzzi et al. analyzed IDE support for collaboration and evaluated an IDE extension to improve team communication [18].

Whereas these studies of information needs focused on existing team structures, our paper introduces a new path for information seeking via on-demand expert support, and the studies present qualitatively different data and implications. Unlike existing team structures, our paper proposes a team structure where a project stakeholder requests remote help from experts who are not stakeholders. This difference has significant implications for team trust, communication preferences, and context sharing.

Collaborative Development

Systems like Codeopticon [16] and Codechella [17] provide ways that helpers (i.e., tutors/peers) can efficiently monitor the behavior of multiple learners and provide proactive on-demand support. Version control systems such as git are often used in programming collaboration because they help developers in distributed teams synchronize source code. However, version control systems also require that developers manually push and pull changes and resolve merge conflicts. Collabode [15] introduced an algorithm that addressed the issue of breaking the collaborative build without introducing the latency and manual overhead of version control. Codeon fetches developers' latest code before helpers can send their code responses back and lets helpers handle all the conflicts if exists.

Communication tools like Slack or Skype make collaboration more effective by supporting conversational interaction, but it is often challenging to capture the code context within these tools. Commercial IDE tools such as Koding [40], and Cloud9 [22] enable users to code collaboratively online in real-time. Although these systems reduce many of the barriers developers face when working at a distance [32] and time spent on environment configuration, they do not support the case when developers are actively seeking help [41]. Codeon allows developers to create requests at any time by speaking their questions while the system automatically captures the problem's context.

IDE-Integrated Help Finding Tools

Codeon is a kind of Recommendation System in Software Engineering (RSSE) [37], which, unlike most CQA websites, often provides relevant information within an IDE. Prior work on RSSEs has used knowledge of how developers seek information to develop systems to provide semi-automated support [10, 11]. Blueprint [9] allows developers to rapidly search for a query in an embedded search engine in their local IDE. Seahawk [34] heuristically filtered search results to automatically increase the reliability of search results within an IDE. Hartmann et al. [19] also explored ways to aid developers in recovering from errors by collecting and mining examples of code changes that fix errors. These in-IDE approaches allow developers to save time by minimizing the change in task context associated with requesting information. Recently, Chen et al. [12] found that even with the state-of-the-art communication tools, such as Skype and JSBin, developers and helpers still face communication challenges when it comes to integrating answers into a codebase.

Tools that support developers using the crowd provide a way to potentially receive more personalized feedback than automated systems can do. CrowdCode [30] allows developers to make requests to the crowd with self-written specifications of the desired function's purpose and signature. But this approach is limited in how much it can reduce developers' time expenditure since making a request requires a detailed problem specification. Real-time crowdsourcing techniques have enabled on-demand interactive systems, which have been shown to be able to improve the efficiency of accomplishing complex tasks [27, 28, 29].

Human Expert Computation

In this paper, we leverage crowdsourcing to make our system available on demand and scalable. By using expert crowd platforms like Upwork [1], which have thousands of developers with a wide range of language and framework expertise, we can hire as many experts as needed to field a developer's set of requests. This allows Codeon to parallelize as much as the end user developer may want to.

Prior work has explored how to use a priori tasking and guidance to automate the coordination and task management process. Foundry [36] provided an interface for composing expert workflows for large tasks. Foundry was used to create *Flash Teams*—dynamic, expert crowd teams—to complete tasks faster and more efficiently than self-organized, or crowd-managed groups. In our work, we focus on similarly-focused tasks with well-scoped hand-offs, but do not assume that developers know the high-level composition of tasks in advance, instead allowing developers to define tasks on-the-fly as they discover and generate them.

CODEON

Codeon's design is based on the feedback we collected over the course of user studies of the three primary stages of help-request interactions: Stage 1) making a request, Stage 2) writing a response, and Stage 3) integrating the response (Figure 1). The design goal per stage is as follows: *(G1)*: to simulate the in-person communication in seeking for help, *(G2)*: to provide ways for a helper to associate responses with the
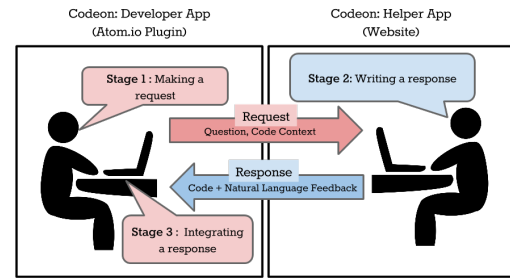
Figure 1: Asynchronous interactions between developers and helpers can occur in three stages: making a request (S1), writing a response (S2), and integrating the response (S3). In Codeon, developers use an IDE plug-in to make requests (S1) and integrate responses (S3), and helpers use a web-based IDE to view content and generate responses (S2).

working code context, and *(G3)*: to make the code integration as effortless as possible

Separating the workflow into three stages makes the communication asynchronous so that a question can be presented to multiple workers and routed to a worker that has right expertise for the scalability reason. These three studies aimed to help us better understand the trade-offs across different methods and features. To minimize the effects of varying prior expertise among participants, all preliminary studies used a synthesized programming language.

Codeon's developer interface is implemented as a plug-in for Atom.io—a widely used code editor—that allows them to make requests (S1) and visualize different formats of responses and integrate responses (S3) within Atom. For helpers, Codeon provides a web IDE that allows them to see a list of developers' requests and respond to them (S2).

Stage 1: Making a Request

User Study: Asking a Good Question

With the primary goal of improving the productivity, we designed this stage with two sub-goals in mind: 1) the speed of request making needs to be fast, and 2) the request needs to contain sufficient context to be understood. With these goals, we compared three modalities for describing requests: 1) speaking the request verbally (Voice), 2) typing the request (Text), and 3) selecting a request from a computed set of categories (Multiple Choice). We derived these categories from common query types observed during a previous study [12].

As prior work showed the importance of context in code request [12], we combined these modalities with three alternative methods for specifying the context of a given request:

Modality	Voice	Text	Multiple Choice
Highlight	11.0 / 2.6	45.5 / 25.4	15.0 / 13.1
Click	14.0 / 3.7	37.5 / 23.1	27.8 / 18.0
None	21.5 / 8.4	33.9 / 17.4	25.1 / 9.5

Table 1: Time to make a request per condition (avg./s.d.)
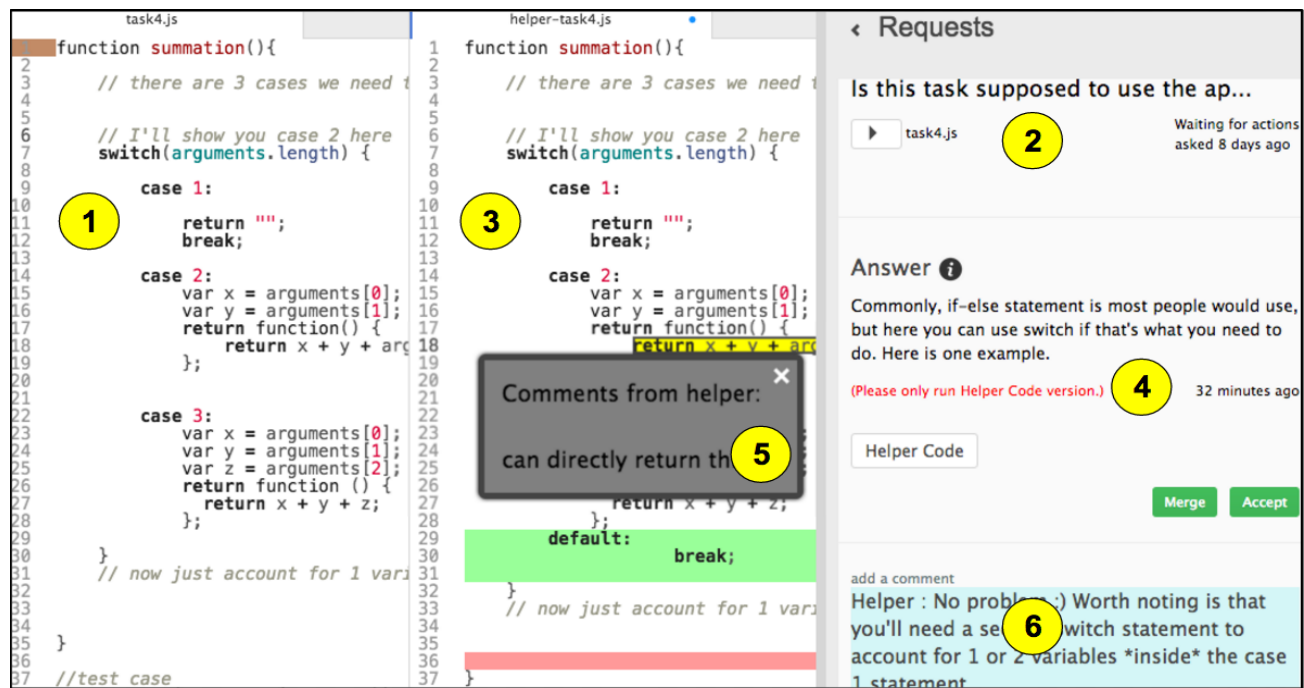
Figure 2: Codeon interface where the requests and responses are on the right panel(2). Developer's code(1) and helper's code(3) are side by side for better comparison. Other responses includes explanation(4), annotation(5), and comments(6)

1) selecting a region of content (Highlight), 2) pointing to one location in the content (Click), and 3) a control condition (No selector). Combining these request modalities and context selectors, we formed a 3×3 condition matrix for our experiment. We recruited 30 workers from Upwork to test the conditions and recorded the duration, content, and user activities of each request.

After removing the unanswerable requests (e.g. those that did not contain enough context), we found that, on average, voice requests were the fastest method for specifying requests, and text input was the slowest (Table 1). In the text condition, participants spent more time carefully crafting requests, whereas in the voice condition participants tended to speak more informally. The performance on the multiple choice option varied based on participants' familiarity with the options. However, we found in a subsequent evaluation of the questions' understandability that the multiple choice specifications were too vague to be understandable by helpers.

Codeon Design: Voicing Requests
As a result of our preliminary studies, we chose to use the speech modality for making requests. When developers make a request, Codeon records their voice, synchronized with their interactions with the editor (typing, highlighting, file switching, and scrolling), which serve as the content selectors. As a request in voice is a dynamic signal, the content selector can also be dynamic so that one request can have an animation of not only the activity of content selection, but also some other informative actions such as typing or viewport changes. This way, a developer can speak and highlight code corresponding to the request, which can be replayed in the helper's interface. This simulates a pair-programming condition where a devel-

oper is asking a question from the person who is co-located by speaking and pointing to content on the screen. In addition, as a result of pilot studies with Codeon, we also added a feature that allows developers to add an optional text title for each request for later reference.

Stage 2: Writing a response
User Study: Response Modalities
In this stage, we want to design features that allow helpers to provide different kinds of response format effectively and efficiently. We conducted a user study with participants responding to a simulated request. One response can have multiple parts and can be written in three different forms: 1) to select a segment of the code and to write an annotation that is associated with the highlighted segment (Code Annotation) 2) to write an explanation in a text box outside of the editor (Explanation), 3) to directly add and modify the code editor (Code Inline), and 4) the combination of all three types (All). We recruited 12 participants and recorded their performance on this task. Based on the common requests that participants had in Stage 1, we created three requests that each participant responded to. We measured the frequency of each answer type and conducted a post-study interview.

Our major high-level finding from this study is that participants' choice of response type depends on the request type, and each response type can well support one or more types of response formats. Overall, we concluded that these three different forms complement each other, and the usage frequency of each type varied based on both the request type and the helper's preference. Table 2 details the trade-offs we found between different response formats in this study in terms effectiveness and efficiency.

Condition	Advantages	Disadvantages	Design Takeaways
Code Annotation	• Preserves original code • Strong connection with the code context	• If the number and the length of annotations increase, they look littered and occlude the main code editor.	• Scalability needed while keeping high visibility
Explanation	• Preserves original code • Better suited for a long conceptual answer	• No connection between code and explanation	• Flexible and accessible
Code Inline	• Quick integration	• Possible merge conflicts • Low visibility	• High visibility necessary

Table 2: Advantages, disadvantages, and design takeaways for three response formats from developers and helpers' perspectives).

Codeon Design - Response Generation

In order to allow helpers to easily view, understand, and respond to each request, the helpers' side of Codeon is built as a web application where a helper can browse a list of developers' requests. Figure 3 illustrates the helpers' web interface. Once specific request is selected, the web application provides a programming environment that shows the files relevant to that request (files that were open in the developer's editor at the time of request generation). As mentioned earlier, a helper is able to not only play the audio that contains the question, but also see the developer's interactions with the Atom editor (e.g., text selection, scrolling, and content editing). Although the request might be involved only part of the original code base, all the scaffolding code are sent along with the request which makes the code executable.

Note that Codeon does not support voice response because we want the system to be scalable such that not only can one worker supports multiple developers but also multiple workers can work on one question. Multiple voice responses will make the response review process time consuming which violates our efficiency design goal. In our study, helpers used all three response types: code annotation, code inline, and explanation. In Codeon, helpers can choose and/or combine different types of answers based on their preference and the question's characteristics.

Stage 3: Integrating a response

User Study: Exploring Response Integration

The last step of the workflow is to review helpers' responses and make changes in the original code. For this step, we want to make sure that the three answer forms (code annotation, explanation, and code inline) can be integrated accurately and quickly by developers. With these two sub-goals in mind, we ran a user study of how developers integrate the same responses presented in different formats. The experiment was composed of four conditions (one condition for each individual response form and one condition with all response forms) with three tasks. We measured the time and accuracy of code integration in each condition.

To make sure a response in different formats contains the equivalent information, we converted an answer in one form to another with specific rules. For example, when converting annotation and code inline to explanation, we specify line numbers to associate the content with specific line numbers of the code. We recruited eight participants with two for each condition, and we recorded their screen during the study, and conducted a post-task interview.

While we cannot generalize the findings due to the sparse number of participants, we can find that explanation took the longest for code integration and the code inline took the shortest. Similar to the conclusion from the second study, the post-interview indicated that there was no strong preference for one of the three types. Rather, participants expressed the trade-off between the types of answers and how each type of answer can be desirable and not desirable in some ways. For example, code annotation is desirable in a way that the textual content can have strong connection with a certain part of the code, complementing the explanation format (which makes it difficult to map the content to the code snippets). Additionally, participants preferred code annotation, as it preserves the helper's original code (and does not add new lines like code inline would). However, it is often seen as being appropriate only for short answers since it is overlaid code editor, meaning that it scales poorly as the number and/or length of annotations increase.

Similarly, explanation format was desirable because it does not corrupt the original code, and it works well for both long and short responses. Compared to annotation and explanation, inline code allowed participants to finish the task more quickly on average. While this is desirable to make the integration process efficient, another challenge is that a participant may miss the code inline added by a helper. This naturally led us to design measures to keep track of a helper's code inline, similar to the Code Diff application which will highlight the new code added. We summarized the advantages and disadvantages for each response format and drew design implications from the results that facilitate future system development (Table 2).

Codeon Design: Response View & Integration

Codeon implements the response panel on the right side of the Atom.io editor. As there can be multiple requests and multiple formats per response, the scalable design is essential. The
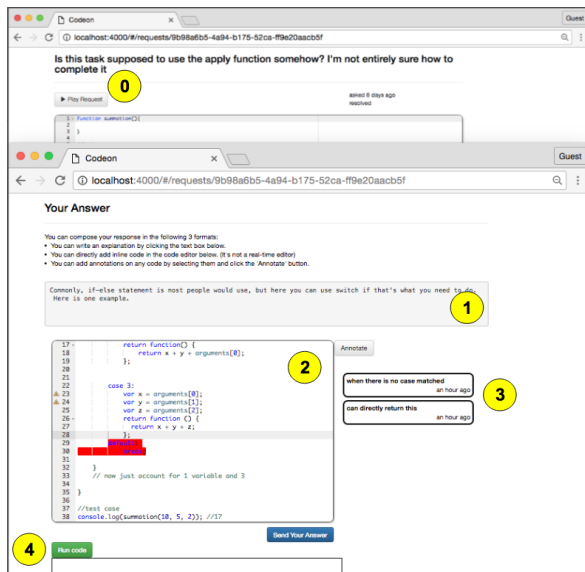
Figure 3: The helper side of Codeon is an interactive webpage that allows helpers to replay the request(0) and run the code(4). Helpers can respond to it with explanation(1), and inline code(2), and annotation(3).

view consists of two hierarchical levels: the requests view and the response detail view (Fig. 2).

The request view has a list of requests where each menu shows the brief summary of the request (title, associated file name, audio replay button) so the developer can keep track of multiple requests. Once a request is selected, the side panel shows the full information of the request and the most recently received response. In addition, if the response contains annotation or inline code, Codeon plugin will automatically split into a two-editor view with the developer's and the helper's code side by side. The region with annotation in the helper's code will be highlighted. When the 'Code Diff' button is clicked, Codeon will display a color-coded difference between the developer's and the helper's code, similar to the 'diff' functionality in modern version control systems (e.g. Git).

Finally, one important goal of the response is to support efficient code integration. With the support of color-coded diff, integration of new code submitted by a helper to the original code can be done in one button click. In addition, for the common issue of the merge conflict in collaborative programming—when more than one person modifies the same content — Codeon is designed to pull the most recent code (if there is any difference) to helper's side by default before helpers sending the responses. This is to reduce the workload of code merge for the developer. Lastly, to support conflict resolution and flexible integration, Codeon generates clear annotated conflict markers for developers, allowing them to automatically merge the helper's code or to restore the original copy.

Iterative Design of the End-To-End System

We iteratively designed the complete Codeon system based on the feedback from 19 developers who used Codeon for a series of small programming tasks. In the initial Codeon version, we found that developers could not 1) efficiently identify the code that responses corresponded to, 2) understand the differences between the original code and the helper's edits, and 3) merge results from helpers into a consistent and functioning solution. The final version of Codeon allows developers to: 1) better connect the content and the request by color mapping the line number and request panel, 2) better integrate helpers' response by refining the code merge strategy, and 3) more easily view and compare to helpers' responses with a side-by-side "Code Diff" tool.

EXPERIMENTAL SETUP

We conducted a laboratory study to better understand how Codeon affects developers' help-seeking behaviors.

Method

Codeon is built for any developers who seek programming support from remote experts. We recruited 12 students from authors' university as developers with the requirement of at least six months JavaScript experience. We also hired three professional programmers as helpers from Upwork (upwork.com), an online freelancer platform, who self-reported multi-year JavaScript experience. The three helpers participated in multiple trials because we found little learning effect in our pilot study and to ensure that the helpers we used met our expertise criteria. We ran an hour and a half training session with each helper to familiarize them with the system and the study. We prepared two JavaScript task sets with each set containing four programming problems. These problems are independent on each other, and their answers cannot be easily found online. To ensure the two sets of tasks were as equally challenging as possible, yet conceptually different, we asked two professional JavaScript developers to balance the tasks. Every developer solved a series of JavaScript tasks in two conditions: a "control" condition and a "Codeon" condition.

In the control condition, developers communicated with helpers via Skype (for real-time synchronous voice communication) and Codepen.io (for real-time synchronous code sharing). The control condition's features are representative of the communication mechanisms that code mentoring sites use [21, 23]. In the Codeon condition, Skype and Codepen.io were disabled, and the developer was instructed to use Codeon to make requests. Both conditions allowed developers to search for online materials. We collected audio and screen recordings during the study to capture the behaviors of the participants, and their responses to our follow-up questions. To minimize learning effects, we randomized the order of conditions (Codeon, control) and the task sets (A, B).

We also instructed participants to finish the tasks as fast as they could by using any resource they were given (online materials and a remote helper), but did not explicitly suggest any strategies. Each study lasted one and a half hour, including training for developers (15 min), the two conditions (30 min per each), and the interview (15 min).

out of 4 programming tasks in 30 minutes, *p-value =0.03*

Figure 4: Overall result: The # of completed tasks is significantly more in Codeon condition(avg./s.d.).

Hypotheses

We designed our study to evaluate four hypotheses:

$H_{\text{Performance}}$: Codeon can help developers to be more productive in development tasks than the control system could.

$H_{\text{SystemTime}}$: Time that developers spend with Codeon to ask for and get help is less than that in the control system.

$H_{\text{Interruptions}}$: Developers get interrupted less often in Codeon than in the control system.

$H_{\text{Parallelization}}$: Developers can better parallelize their efforts in Codeon than they can in the control system.

RESULTS

Overall Performance

The productivity of each condition was measured by counting the number of completed tasks (out of four tasks per condition given 30 minutes cutoff time). Figure 4 shows that the average number of completed tasks within the given time in the Codeon condition is significantly more than it is in the control condition (two-tailed paired-samples Students T-Test, $p = 0.03$), which supports our hypothesis $H_{\text{Performance}}$. To understand *why* developers were more effective with Codeon, we further analyzed our user data and described in the following sections.

Individual Task Performance

To understand the advantages of Codeon, we unpacked our data to investigate participants' performance on each task. As Table 3 shows, participants spent less time in completing task

	Codeon	Control	Time Increase(%)
Time spent per completed task	10.57	11.32	7.1%
Time spent per incomplete task	8.49	9.15	7.8%
Time spent per incomplete task in non-tail condition	6.84	8.47	23.7%

Table 3: The average time (in minutes) spent per task. Participants spent longer in the control condition, on both completed and incomplete tasks. Tasks in tail condition is the task that are stopped by the researchers by the time constraints (30 min). Note that, by definition, there cannot be complete task in the tail condition.

name	Codeon	Control	p-value
Avg. # of requests per completed task	1.71(1.41)	2.18(1.54)	0.45
system active time(sec) per completed task	165.8(106.3)	344.4(249.5)	0.05

Table 4: The # of requests made per completed task is not significantly different. The average system active time per completed task in Codeon is longer than in the control condition (avg./s.d.).

on average when using Codeon condition, although the difference is not significant. While Codeon may help developers to complete task quicker than the control model, the difference is not significant enough to account for outperforming. Meanwhile developers may waste more time on a task when they get stuck with it in the control condition. The time spent per incomplete task also support this conjecture. Especially if we exclude the incomplete tasks that were stopped by the researchers for 30 minutes ("tail condition"), we can observe a 23% time increase in the control group. As the time spent on the incomplete tasks were determined by an external factor (the time constraint), not by the developer's intention, we believe this measure better reflects the time spent on incomplete tasks for the comparison purpose. While we cannot calculate the statistical significance for these three measures as samples in each condition are part of the entire data set (e.g. complete tasks in Codeon are different from the complete tasks in the control condition), the results indicate that the improvement in overall productivity potentially comes from wasting less time when the developer cannot solve the problem in Codeon. We hypothesize ($H_{\text{Parallelization}}$) that the asynchronous nature of Codeon workflow encourages developers to hand off their work to a helper and to move on to the next task, whereas, in the pair-programming session, two developers typically work on the same task at a time. In the next section, we evaluate if developers parallelize work efforts during the experiments.

As we do not find strong evidence in developer completing tasks faster in Codeon, we further analyze how actively developers utilize the assistance system when they were able to complete tasks so as to give an account of the increase in overall performance. The average number of requests and *system active time* per complete task is reported in Table 4. System active time is the time that a developer spent on the assistance system (Codeon or the control system) to make requests to a helper and to receive assistance from the helper. System active time thus includes any time that would not have been needed if there was no helper, for example, watching the helper programming (in CodePen), creating a request, reviewing responses from the helper, or interacting with the helper (via Skype, CodePen or Codeon). The result shows we cannot see a significant difference in the number of requests made per complete task ($p = 0.45$). The system active time per completed task in Codeon, on the other hand, is less than the one in the control condition ($p = 0.05$), which shows a tendency to significance for our hypothesis $H_{\text{SystemTime}}$. Based on our self-assessment from the video annotation process, we notice that the cost of extra time in the control condition may come from the nature of synchronous communication

name	Codeon	Control
Avg. # of alerts	6.1(3.0)	1.9(2.2)
Avg. # of interruptions	2.5(1.6)	1.9(2.2)
Ave. of interruption/alert	0.48(0.3)	1.0(0)

Table 5: # of interruptions, alerts, and average of individual ratio of interruption/alert(Avg. / S.D.).

name	Codeon	Control
Avg. # of parallelization per developer	2.1(1.2)	0.3(0.6)
Total time(s) of parallelization	281.9(243.5)	2.4(5.7)
Avg. time(s) of parallelization per occurrence	114.2(80.5)	1.9(4.8)

Table 6: Time spent and the number of parallelization behavior in two conditions (Avg. / S.D.).

between two ends. For example, a remote pair-programming session may cost additional time coming from social norms, real-time typing process, additional out-of-context questions (or feedback) [13] that may not contribute to the overall performance and does not exist in the Codeon model. This aligns with our belief that Codeon is more efficient in seeking for and receiving help from a remote assistant. The efficiency in Codeon can potentially cause an overall increase in the performance by expediting completion time or giving the developers more time to complete.

Interruptions and Parallelization
Studies have shown that interruptions can be costly to programmers [24]. As Codeon follows the asynchronous collaboration model, we analyze the occurrences of a helper interrupting a developer and evaluate if it has any advantage of being less disruptive to the developers. Annotating the screen recordings of each experiment, we count the number of alerts and interruptions. An *alert* is a message from a helper that initiates a conversation, which gets an attention from the developer or notifies the developer that a response/comment is received. Receiving an alert does not necessarily mean that the developer needs to take action immediately or is interrupted. For example, in Codeon, a developer can see the notification of a helper's response and review the response later, once the work being carried out is done, or, in the control condition, the developer can ask the helper to wait a little while. In addition, the task that the developer was currently working on may be directly relevant to what the helper responded so that the interrupted task not needed to be resumed. We say an alert causes an *interruption* if the two following conditions are satisfied: i) the alert makes the developer immediately stop what they are working on in order to review or respond to the helper's message, and ii) the stopped task needed to be resumed later. Table 5 shows the absolute numbers of both alerts and interruptions are greater in the Codeon condition. This is because in Codeon, one comment is counted as an alert, whereas in the control condition, the developer and the helper constantly communicate so that they have a smaller chance to be interrupted as they are working together. However, when a helper alerts a developer in the conference call, the developer has to stop the current task 100% of the time. In the meantime, in Codeon, they were interrupted (immediately respond to the helper and later resumed the task) only half of the time (48%), and otherwise they could keep working on their task until the point that they finish the current activity (e.g. finishing the line that was being written, finishing reading online materials that were being read). Even when developers were interrupted in Codeon, we observed that most of the interruptions did not require a significant context switch in the developer's mental model as the interrupted task was

relevant to the response from the helper. We did not choose to evaluate this as it can be subjective. If we look further detail for individual, 5 out of 12 developers chose to wait to review responses and, on average, they spent 18.1 seconds to finish the ongoing activity. Potentially, this tendency can scale once the system is deployed and is constantly used by developers. Indeed, using Codeon, developers can have better control over their workflow by having a smaller number of interruptions ($H_{Interruptions}$) whereas, in synchronous collaboration, the workflow will be determined by the pair otherwise the developer will be interrupted.

As briefly mentioned, another benefit of asynchronous collaboration can come from a developer parallelizing the task by handing off subtasks to helpers. To confirm the possible benefit in Codeon, we annotated the video to see if developers parallelize their work while waiting for responses from helpers. We present the number of parallelization and the time that the developers parallelize their tasks in Table 6. Table 6 presents that developers parallelized their work 2.1 times on average when they hand off their work to the helper ($mean = 2.1$) in Codeon condition, whereas in the control condition this behavior occurred close to zero ($mean = 0.3$). In addition, their time spent on parallelization is much longer in the Codeon condition. Furthermore, the two developers with parallelization behavior in control condition were instantly interrupted ($mean = 1.9s$) by helpers when they attempted to work on different tasks. We found that 11 out of 12 developers parallelized their work when using Codeon, but only two when using the control system. Thus we can assume the parallelization is natural in the setting of Codeon. The result supports our hypothesis $H_{Parallelization}$ that Codeon supports the distributed workflow. This potentially account for the improvement performance.The evidence and analysis above provide us with insights on the overall performance of two systems. Next, we review developers' feedback and screen recording to facilitate the qualitative analysis.

Post Interview and Developer Feedback
Parallelization
Nearly every developer (11/12) parallelized their efforts. We discovered two patterns of parallelization behavior from both post-interviews as well as our observations. After sending a request or comment, developers would either 1) review a different task, or 2) work on another part of the same task. The first pattern is more common, and some developers used it directly after they read the problem. The second pattern often happened in those tasks with multiple requirements. Developers would divide the task into a few subtasks and distribute some to helpers. For example, one task asks to remove the duplicates of an array and then sort it. One developer (P4) asked

his helper to write a function to remove the duplicates. While waiting for a response, he started to code the sort method. Another developer (P9) moved on to a search task after making requests about writing a method and code debugging.

"I was able to kinda break down the tasks into subtasks, and kind of, things I can ask him to help with, and things I can work myself. (P9)"

In general, we observed that developers consistently showed a tendency to parallelize their work, regardless of the condition. However, we found that Codeon allows them to accomplish the distributed workflow.

Interruption
The post-interview also supports our hypothesis $H_{Interruptions}$ by having five participants directly mention the interruption issues in the control system (none in Codeon). There are two types of interruptions we noticed. One is direct interruption which we defined in the previous section, and the other is more subtle distraction coming from the conference call itself. For example,

"When using Skype, he kept asking me about clarifying things that I asked him, I couldn't do anything at the same time, like I had to pay my attention to what he's asking and make sure that whatever I'm asking him, he understood properly. (P9)"

Three developers in the control condition, although working on other tasks while waiting for helpers, were interrupted by their helpers (e.g. asking for confirmation, requesting to check for answers). The helper regularly asked for confirmation such as "you see this?", which force the developer to switch applications back and forth to interact with the helper until they eventually decided to solve this problem together.

CODEON FOR NON-PARALLELIZABLE TASKS
Our results demonstrate that Codeon can help developers complete more tasks when there is the possibility of parallelizing tasks. However, there may remain a subtle but important trade-off: if the success of Codeon is contingent on the amount of parallelism possible for a given problem, then there may be a "minimum" amount of task parallelism below which the baseline condition outperforms Codeon. Intuitively, predicting how parallelizable a future request will be in order to select the most effective system is highly impractical for developers. Fortunately, we found no evidence that this is ever necessary with Codeon.

To test if there exist any such cases where the baseline outperforms Codeon, we ran a study with 14 pairs of programmers (separate from those used in our primary study). We chose the hardest possible case for Codeon: program with an unfamiliar language with no potential parallelism between tasks. To make parallel tasks infeasible, developers were required to have no experience with the language used in the task (AngularJS). Because of unfamiliarity and all the tasks required some levels of understanding to finish, real time communication would provide stronger support.

Even comparing to an earlier version of Codeon, we found no evidence that the control condition outperformed Codeon ($p > 0.50$), even in this most challenging scenario. While

we cannot conclusively prove these two cases are not different (i.e., disprove the null hypothesis), the lack of evidence of a difference in this study, and our anecdotal experience observing participants suggests no reason to expect a trade-off between the two systems to exist. Therefore, our data shows that Codeon can perform at least as well as the control system.

Additionally, we observed that 74.17% of the time a developer was engaged with the helper on average, versus 44% with Codeon (mean=0.44, s.d.=0.158, $p < 0.0005$). This further affirms the efficiency observations in our results.

DISCUSSION
Codeon User Interface
Almost all the developers (11/12) in the experiment gave positive feedback on Codeon's user interface, that supports our design decisions retrieved from the series of user studies.

For example, participants mentioned that making requests by using multimodal interaction (voice + code context) allows them to "give context easier" (P4). Furthermore, participants were generally in favor of the way in which Codeon integrates code-based responses. For example, the pop-up notification and the alert sound coming with the new message helped developers to be notified more quickly.

"..the notifications that it gave me are very good... comparing to only have text, add sound can help me to... when i look at the left i can still know what happens on the right.(P5)"

Codeon also prevents developers from "missing something that a helper wrote", and helps them better understand the code by allowing them to "compare two code files simultaneously" (P7). Two developers mentioned that they could not follow where the helper was typing in the control condition because, unlike Codeon, the interactions cannot be replayed nor easily recorded.

"In Codepen, the helper is changing in another window (other than Atom) that I have no idea what he did. In Skype, if I have a question I just say it but there is no history. (P3)"

Effects of Social Norms
Previous research shows that lower social burden on asynchronous communication activity than synchronous [4]. We also found that developers in the control condition expressed the challenge in real-time communication: phrasing the requests, or explaining the code. With the study setup of having a remote helper available in real-time via a conference call, four developers addressed that they were less comfortable and felt more pressure because they felt they "have to ask something" (P11), and less comfortable when having someone just "sitting there" (P11, P4). The rest of them felt little pressure and relied on helpers more to solve the problem. On the other hand, no one expressed similar concerns for Codeon. Codeon offers a more independent environment with little social pressure and the full control over code and the assistance pipeline.

"In Skype, but I also felt like, not that he's interrupting me, but like I can just hear him in the background, its kind of, not intimidating, but like, make me feel like I had to ask questions, even though I wanted to do stuff on my own. (P11)"

Potential of Codeon

One of the limitations of our study is that there are only four tasks, which limits Codeon's potential to support parallelization better and may lead to a larger difference in productivity. For example, there is one developer finished all four tasks in the Codeon condition within 30 minutes. Also, we found that the developers, who only have one task left before the session ended, cannot parallelize their effort(as they could before) because there are no other tasks left. Instead, after making a request for one task, they would continue working on that same task which creates redundancy.

In contrast, many developers expressed concerns on the control system for the synchronous nature of the collaboration. For example, as two developers are sharing an editor, the editor can be a limited resource that blocks a developer's interaction. Participants expressed that they felt limited for various reasons: being "stuck watching" (P12) the helper's typing, being distracted with the other "hear him in the background" (P11), or "delete my code and directly add his code" (P7). Especially given the social barrier in which a developer is put to work with a remote stranger, the pair programming model revealed some challenges for developers in engaging with the helpers in a short time.

Generalizing Codeon's Approach

Our target audiences are programmers who need support that can be more efficiently provided by remote expert developers than existing methods. This remote assistance model can be useful in many contexts including education and distributed teams. We focus on developers' communication, which is important in all of these use cases, regardless of team size or incentive. The three stages of Codeon system we have discussed before fit into a more general model that Codeon advances. Specifically, tools for generating sub-tasks in the current context of work (S1), making it easy for helpers to 're-hydrate' that context (S2), and providing tools for quickly and effectively integrating contributions (S3), can be used across various domains, for example, using software made for professionals (e.g. Photoshop or Final Cut Pro) Exploring how to effectively recreate this approach in other settings is future work beyond the scope of this paper.

LIMITATIONS AND FUTURE WORK

Codeon is an early step towards always-available, crowd-powered developer assistance. As such, there are several limitations to the current system, but also many exciting directions of future work that we hope to bring light to in the HCI, software engineering, and crowdsourcing communities.

Input Modalities

Based on observations and feedback, we found that more than half of participants prefer to have some of their requests made only in text. This is either because they are "used to typing questions" (P6), or they feel it is "hard to speak their questions clearly with one recording", or it is just because "typing is more convenient"(P6) for some cases. Indeed, three developers phrased their requests in the request title and did not record any audio. This suggests that we could enable both text and voice request modalities in future systems.

New Hiring Models for Expert Crowds

On-demand hiring models in prior work have mostly focused on non-expert workers (e.g. [8, 7]). These models assume a workflow that does not validate specific expertise, and usually involve a posted request that includes instructions in details. Future work will explore methods for ad hoc team formation and new expert-sensitive recruiting strategies, and will be able to leverage Codeon as a platform for testing these methods in the wild.

Team-Based Support of Requests

Not only may we be able to recruit individual expert helpers to field requests in parallel, but also we can begin exploring how teams can be formed around these tasks. This line of work shares many of the motivations of the prior work on Flash Teams [36]—quick assembly of efficient, scalable expert teams—but also aims to accomplish this without a priori knowledge of task structure. In place of a priori knowledge, we may be able to leverage what the system can understand about the structure and interplay of the user's existing codebase to find subtasks and even critical skill sets that may be hard to find individual expert helpers to completely address. We believe that Codeon provides an ideal platform for research such as this.

Longitudinal Deployment Studies

Using new tools in the software development process (or any expert workflow) requires time to acclimate to observe the true final effect. People become more comfortable with the tool, more knowledgeable in how to best use it in their work, and begin to plan out their tasks in the context of having the tool at their disposal. While our initial results in this paper show tremendous promise for Codeon, future work will study how developers' processes (both individual and collaborative) change with long-term use of the tool in team and for-hire settings. We are already starting to partner with development organizations, but conducting such a long-term evaluation is beyond the scope of this paper.

CONCLUSION

In this paper, we introduced Codeon, an in-IDE tool that allows software developers to get asynchronous on-demand assistance from remote programmers with minimal effort. Our results showed that developers using Codeon are able to complete nearly twice as many tasks as they could using state-of-the-art synchronous video and code sharing tools, by reducing the coordination costs of seeking assistance from other developers. We have already begun using Codeon in our research group to get external help, as well as efficiently collaborate within our own teams. In the future, in-IDE assistance can be used to further improve productivity, reduce interruptions, and even leverage a combination of human and machine intelligence to aid developers.

REFERENCES

1. Upwork inc. (formerly odesk), https://www.upwork.com, 2015. Accessed: April, 2016.

2. Ackerman, M. S. Augmenting organizational memory: a field study of answer garden. *ACM Transactions on Information Systems (TOIS) 16*, 3 (1998), 203–224.

3. Ackerman, M. S., and McDonald, D. W. Answer garden 2: merging organizational memory with collaborative help. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, ACM (1996), 97–105.

4. Almaatouq, A., Alhasoun, F., Campari, R., and Alfaris, A. The influence of social norms on synchronous versus asynchronous communication technologies. In *Proceedings of the 1st ACM international workshop on Personal data meets distributed multimedia*, ACM (2013), 39–42.

5. Asaduzzaman, M., Mashiyat, A. S., Roy, C. K., and Schneider, K. A. Answering questions about unanswered questions of stack overflow. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, IEEE Press (2013), 97–100.

6. Baheti, P., Gehringer, E., and Stotts, D. Exploring the efficacy of distributed pair programming. In *Extreme Programming and Agile MethodsXP/Agile Universe 2002*. Springer, 2002, 208–220.

7. Bernstein, M. S., Brandt, J., Miller, R. C., and Karger, D. R. Crowds in two seconds: Enabling realtime crowd-powered interfaces. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, ACM (2011), 33–42.

8. Bigham, J. P., Jayant, C., Ji, H., Little, G., Miller, A., Miller, R. C., Miller, R., Tatarowicz, A., White, B., White, S., et al. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, ACM (2010), 333–342.

9. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S. R. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2010), 513–522.

10. Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., and Klemmer, S. R. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2009), 1589–1598.

11. Brandt, J., Guo, P. J., Lewenstein, J., Klemmer, S. R., and Dontcheva, M. Writing code to prototype, ideate, and discover. *Software, IEEE 26*, 5 (2009), 18–24.

12. Chen, Y., Oney, S., and Lasecki, W. Towards providing on-demand expert support for software developers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2016).

13. Chong, J., and Hurlbutt, T. The social dynamics of pair programming. In *29th International Conference on Software Engineering (ICSE'07)*, IEEE (2007), 354–363.

14. Cockburn, A., and Williams, L. The costs and benefits of pair programming. *Extreme programming examined* (2000), 223–247.

15. Goldman, M., Little, G., and Miller, R. C. Real-time collaborative coding in a web ide. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, ACM (2011), 155–164.

16. Guo, P. J. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th annual ACM symposium on User interface software and technology*, ACM (2015).

17. Guo, P. J., White, J., and Zanelatto, R. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*, IEEE (2015).

18. Guzzi, A., Bacchelli, A., Riche, Y., and van Deursen, A. Supporting developers' coordination in the ide. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, ACM (2015), 518–532.

19. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. R. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2010), 1019–1028.

20. Herbsleb, J. D., Klein, H., Olson, G. M., Brunner, H., Olson, J. S., and Harding, J. Object-oriented analysis and design in software project teams. *Human–Computer Interaction 10*, 2-3 (1995), 249–292.

21. Inc., C. Code mentor, https://codementor.io/, 2014. Accessed: April, 2016.

22. Inc., C. I. Cloud9 ide, https://c9.io, 2010. Accessed: April, 2016.

23. Inc, H. Hack.hands(), https://hackhands.com/, 2015. Accessed: April, 2016.

24. Iqbal, S. T., and Horvitz, E. Disruption and recovery of computing tasks: field study, analysis, and directions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2007), 677–686.

25. Ko, A. J., DeLine, R., and Venolia, G. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, IEEE Computer Society (2007), 344–353.

26. Ko, A. J., Myers, B., Aung, H. H., et al. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, IEEE (2004), 199–206.

27. Lasecki, W., Miller, C., Sadilek, A., Abumoussa, A., Borrello, D., Kushalnagar, R., and Bigham, J. Real-time captioning by groups of non-experts. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, ACM (2012), 23–34.

28. Lasecki, W. S., Kim, J., Rafter, N., Sen, O., Bigham, J. P., and Bernstein, M. S. Apparition: Crowdsourced user interfaces that come to life as you sketch them. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, ACM (2015), 1925–1934.

29. Lasecki, W. S., Wesley, R., Nichols, J., Kulkarni, A., Allen, J. F., and Bigham, J. P. Chorus: a crowd-powered conversational assistant. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, ACM (2013), 151–162.

30. LaToza, T. D., Towne, W. B., Adriano, C. M., and van der Hoek, A. Microtask programming: Building software with a crowd. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, ACM (2014), 43–54.

31. LaToza, T. D., Venolia, G., and DeLine, R. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ACM (2006), 492–501.

32. Olson, G. M., and Olson, J. S. Distance matters. *Human-computer interaction 15*, 2 (2000), 139–178.

33. Overflow, S. Stack overflow, https://stackoverflow.com/, 2015. Accessed: April, 2016.

34. Ponzanelli, L., Bacchelli, A., and Lanza, M. Seahawk: Stack overflow in the ide. In *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press (2013), 1295–1298.

35. Raymond, E. S. *The Cathedral and the Bazaar*, 1st ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

36. Retelny, D., Robaszkiewicz, S., To, A., Lasecki, W. S., Patel, J., Rahmati, N., Doshi, T., Valentine, M., and Bernstein, M. S. Expert crowdsourcing with flash teams. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, ACM (2014), 75–85.

37. Robillard, M. P., Walker, R. J., and Zimmermann, T. Recommendation systems for software engineering. *Software, IEEE 27*, 4 (2010), 80–86.

38. Schenk, J., Prechelt, L., and Salinger, S. Distributed-pair programming can work well and is not just distributed pair-programming. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ACM (2014), 74–83.

39. Sillito, J., Murphy, G. C., and De Volder, K. Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on 34*, 4 (2008), 434–451.

40. Sinan Yasar, D. Y. Koding, 2012. Accessed: April, 2016.

41. Steinmacher, I., Silva, M. A. G., and Gerosa, M. A. Barriers faced by newcomers to open source projects: a systematic review. In *Open Source Software: Mobile Open Source Technologies*. Springer, 2014, 153–163.